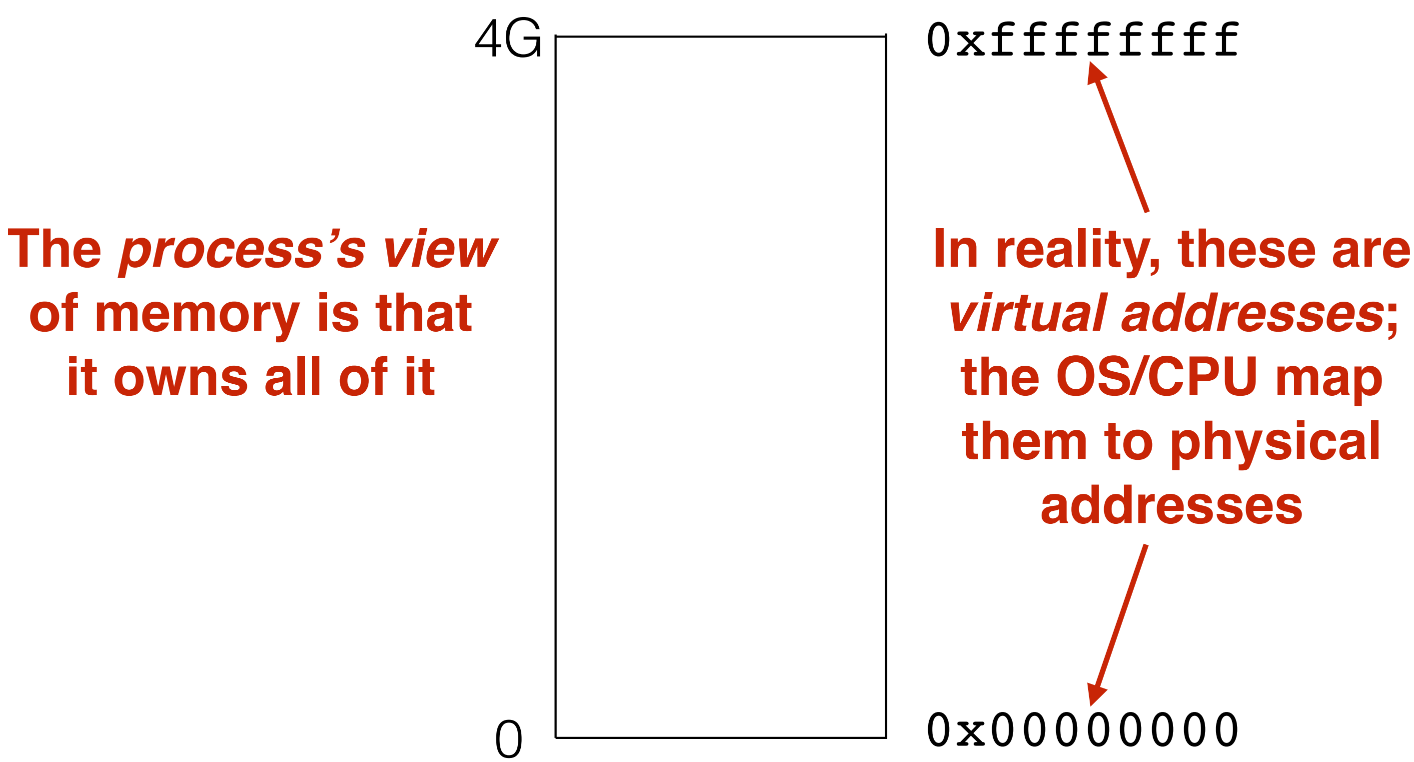
Memory layout

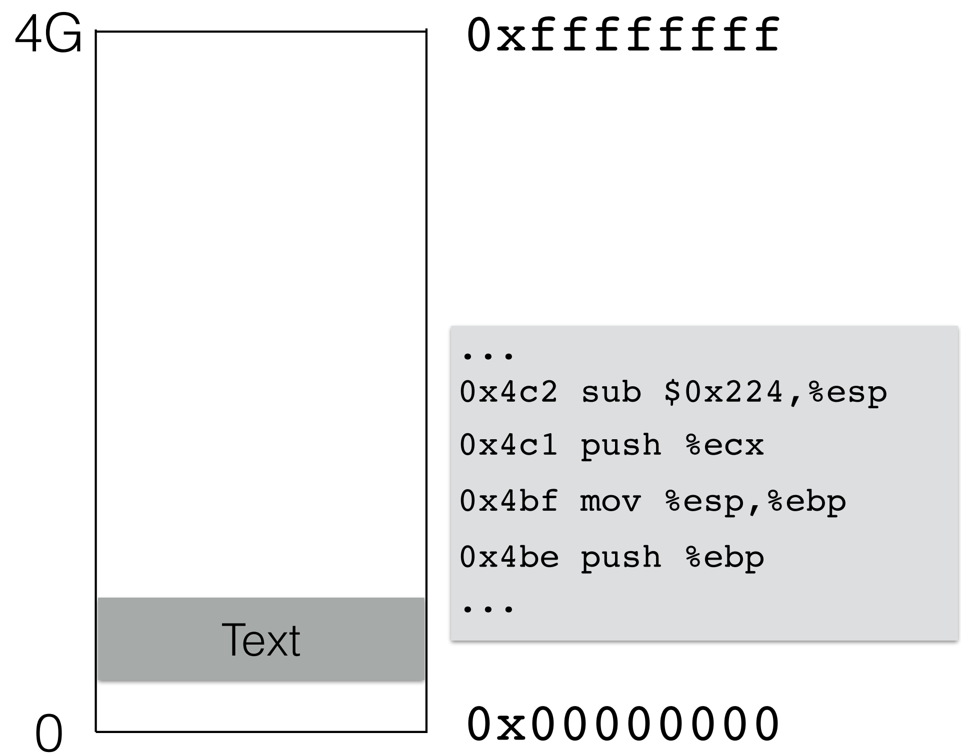
Memory Layout Refresher

- How is program data laid out in memory?
- What does the stack look like?
- What effect does calling (and returning from) a function have on memory?
- We are focusing on the Linux process model
    - Similar to other operating systems

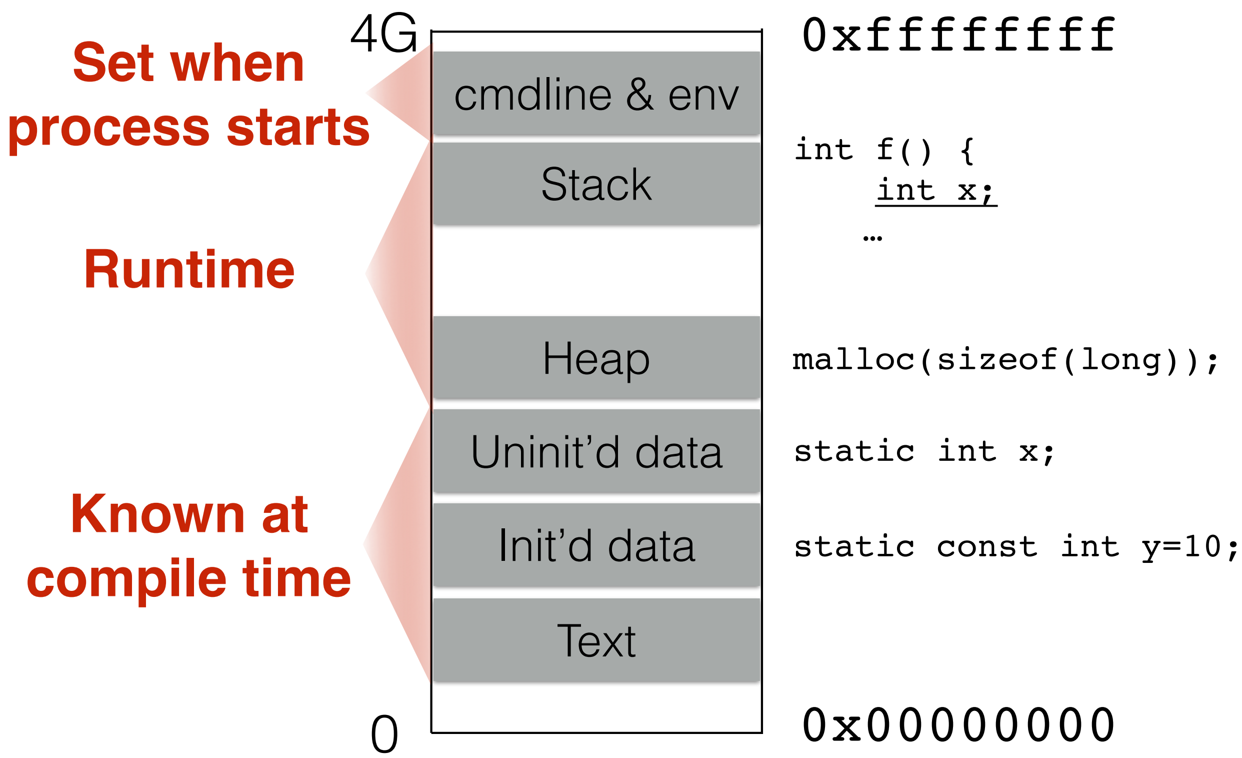
All programs are stored in memory



The instructions themselves are in memory



Location of data areas

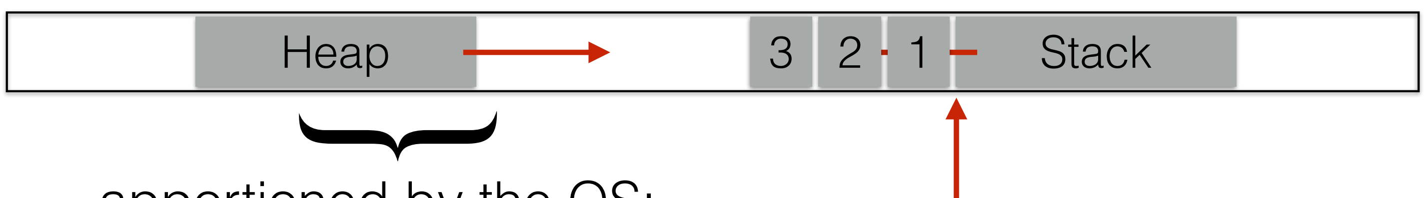


Memory allocation

Stack and heap grow in opposite directions

Compiler emits instructions
adjust the size of the stack at run-time

`0x00000000` `0xffffffff`



apportioned by the OS;
managed in-process
by **`malloc`**

Stack
pointer

```
push 1
push 2
push 3
return
```

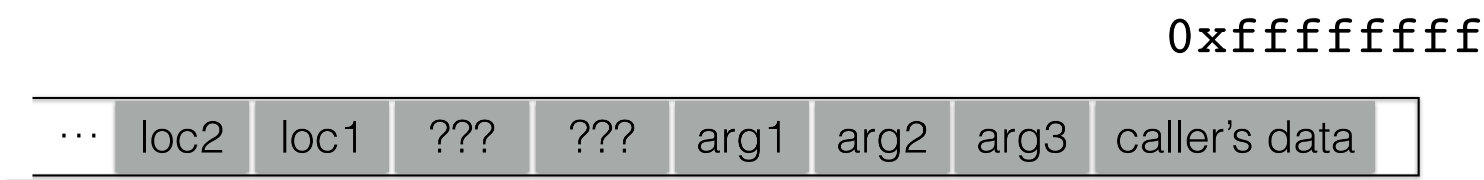
Focusing on the stack for now

Stack and function calls

- What happens when we **call** a function?
  - What data needs to be stored?
  - Where does it go?
- What happens when we **return** from a function?
  - What data needs to be *restored*?
  - Where does it come from?

Basic stack layout

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```



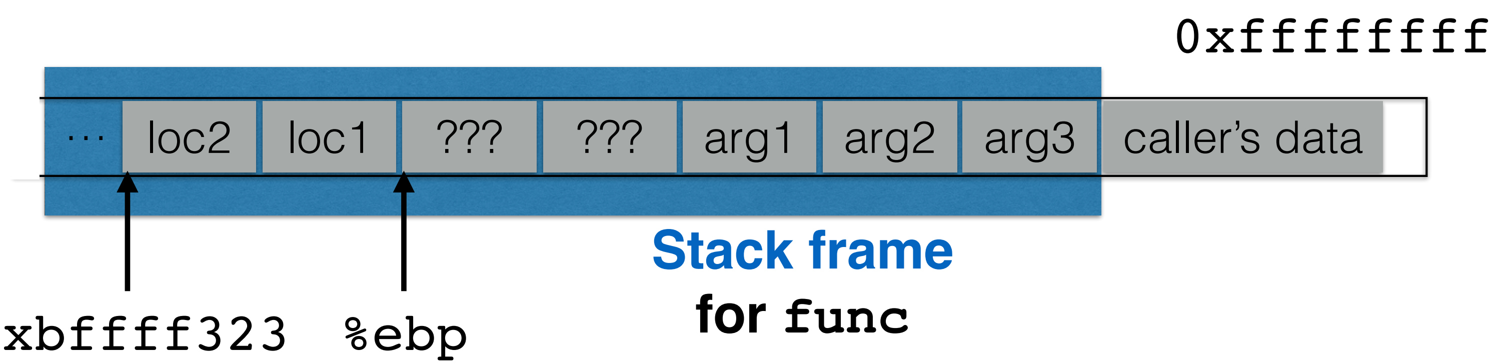
Local variables pushed in the same order as they appear in the code

Arguments pushed in reverse order of code

Accessing variables

```
void func(char *arg1, int arg2, int arg3)
{
    ...
    loc2++;
    ...
}
```

Q: Where is (this) `loc2`?
A: -8(%ebp)



Can't know absolute address at compile time

Frame pointer

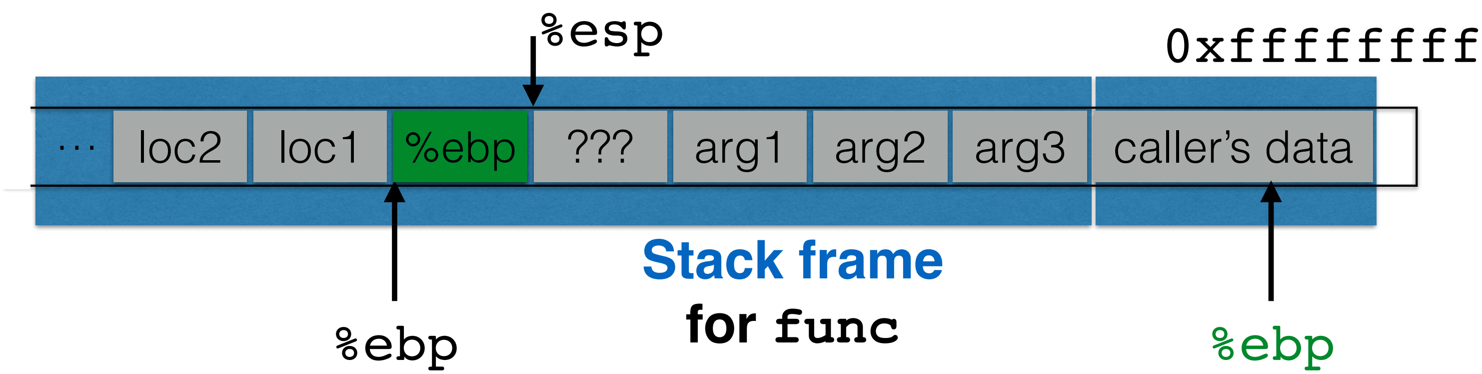
But can know the **relative** address

- **`loc2`** is always 8B before ???s

Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we restore %ebp?



Push `%ebp` before `locals`

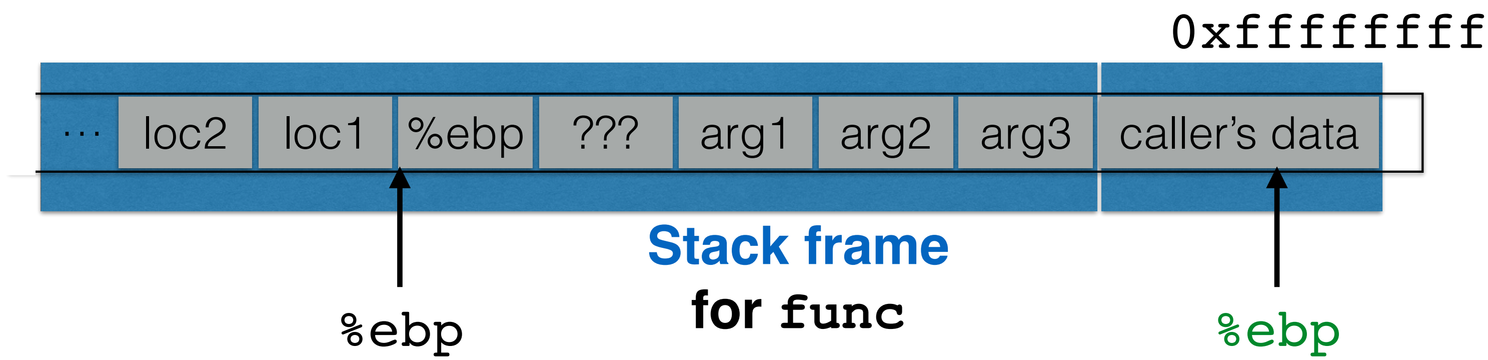
Set %ebp to current (%esp)

Set `%ebp` `to(%ebp)` at return

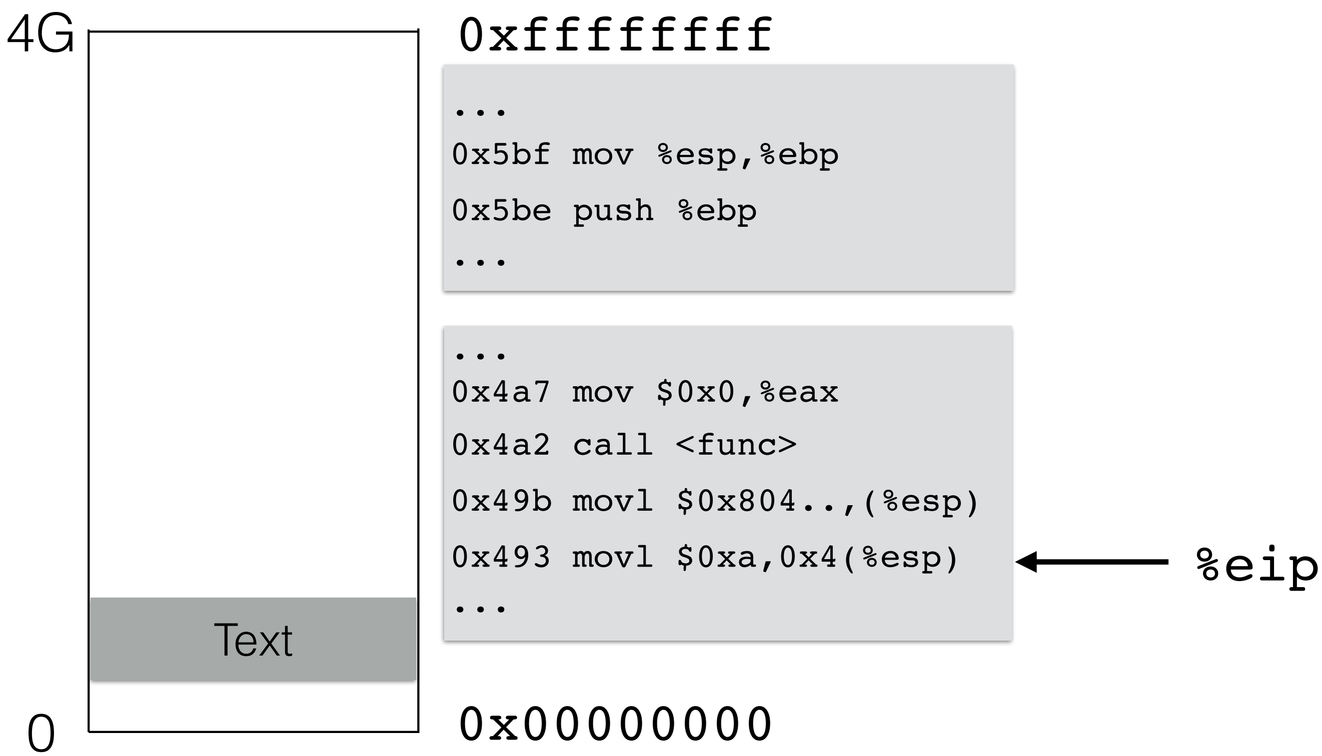
Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we resume here?



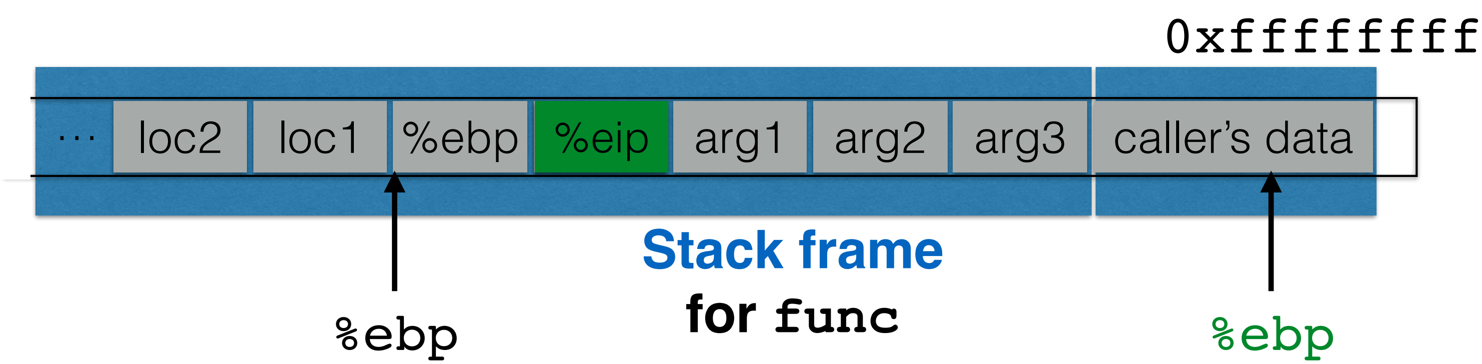
Instructions in memory



Returning from functions

```
int main()
{
    ...
    func("Hey", 10, -3);
    ...
}
```

Q: How do we resume here?



Set `%eip to 4(%ebp)` at return

Push next `%eip` before `call`

Stack and functions: Summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump to the function's address**

Called function:

4. **Push the old frame pointer** onto the stack (%ebp)
5. **Set frame pointer** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)

Buffer overflows

Buffer overflows from 10,000 ft

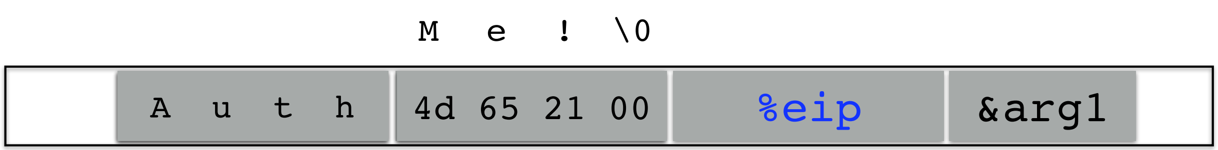
- **Buffer** =
  - Contiguous memory associated with a variable or field
  - Common in C
    - All strings are (NUL-terminated) arrays of `char`'s
- **Overflow** =
  - Put more into the buffer than it can hold
- **Where does the overflowing data go?**
  - Well, now that you are an expert in memory layouts…

Benign outcome

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, str);
    ...
}

int main()
{
    char *mystr = “AuthMe!”;
    func(mystr);
    ...
}
```

Upon return, sets `%ebp` to `0x0021654d`



buffer

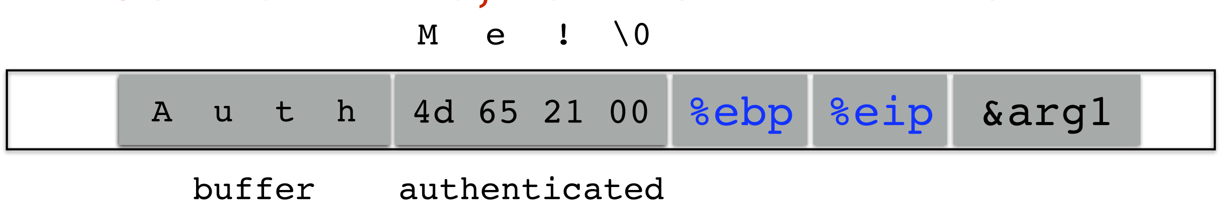
SEGFAULT (0x00216551)

Security-relevant outcome

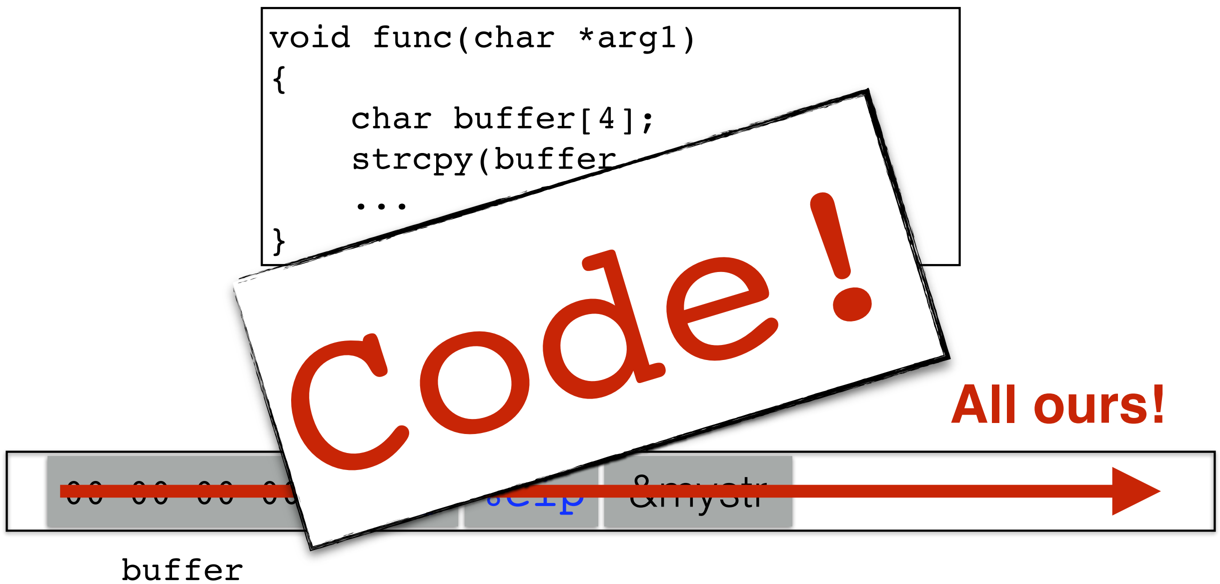
```
void func(char *arg1)
{
    int authenticated = 0;
    char buffer[4];
    strcpy(buffer, str);
    if(authenticated) { ...
}

int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

Code still runs; user now ‘authenticated’



Could it be worse?



`strcpy` will let you write as much as you want (til a ‘\0’)

What could you write to memory to wreak havoc?

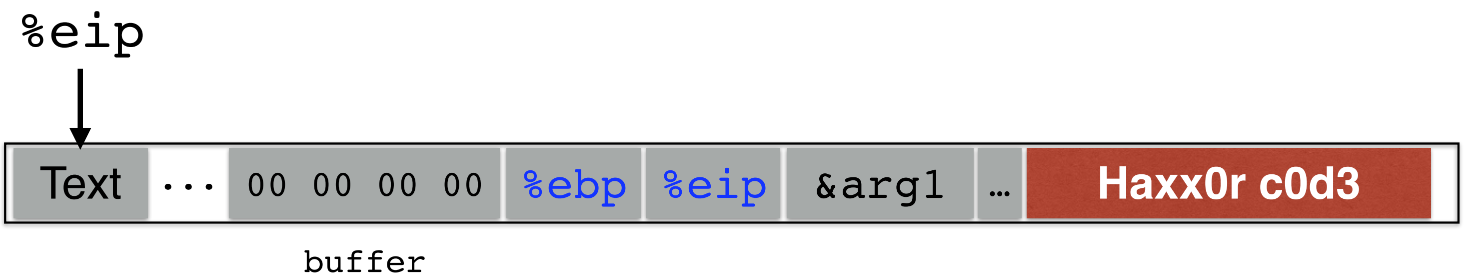
Aside: User-supplied strings

- These examples provide their own strings
- In reality **strings** come **from *users*** in myriad aways
  - **Text** input
  - **Packets**
  - **Environment variables**
  - **File** input…
- **Validating assumptions** about **user input** is extremely **important**
  - We will discuss it later, and throughout the course

Code injection

Code Injection: Main idea

```
void func(char *arg1)
{
    char buffer[4];
    sprintf(buffer, arg1);
    ...
}
```



(1) Load my own code into memory

(2) Somehow get `%eip` to point to it

Challenge 1
Loading code into memory

- It **must be the machine code** instructions (i.e., already compiled and ready to run)
- We have to be careful in how we construct it:
  - It **can't contain** any **all-zero bytes**
    - Otherwise, sprintf / gets / scanf / … will stop copying
    - How could you write assembly to never contain a full zero byte?
  - It **can't use the loader** (we're injecting)
  - It **can't use the stack** (we're going to smash it)

What code to run?

- Goal: **general-purpose shell**
  - Command-line prompt that gives attacker **general access to the system**
- The code to launch a shell is called **shellcode**

Shellcode

```c
#include <stdio.h>
int main( ) {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

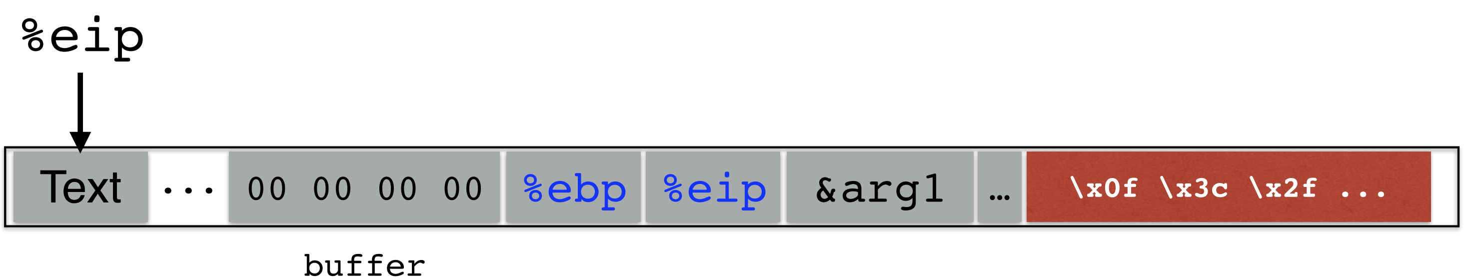
Machine code

(Part of) your input

Challenge 2

Getting injected code to run

- We can't insert a "jump into my code" instruction
- We don't know precisely where our code is



Recall

Memory layout summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump to the function's address**

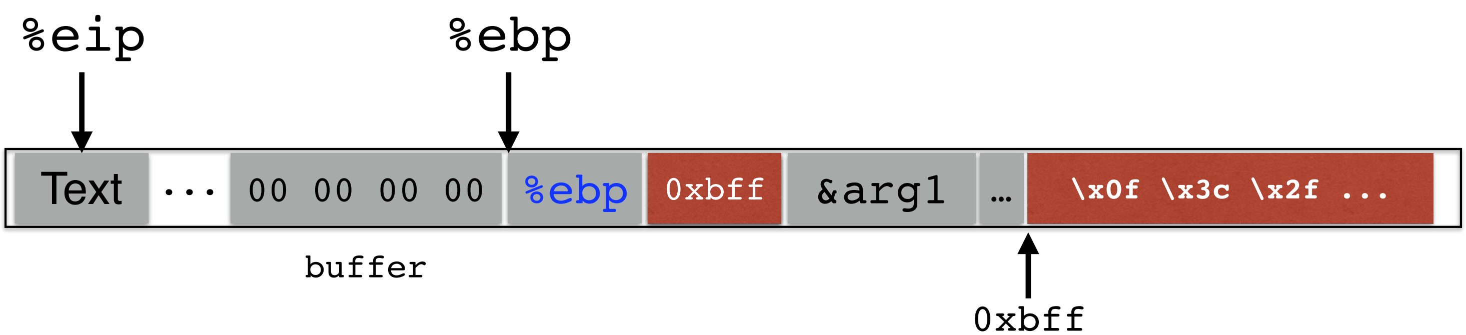
Called function:

4. **Push the old frame pointer** onto the stack (%ebp)
5. **Set frame pointer** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

7. **Reset the previous stack frame**: %ebp = (%ebp)
8. **Jump back to return address**: %eip = 4(%ebp)

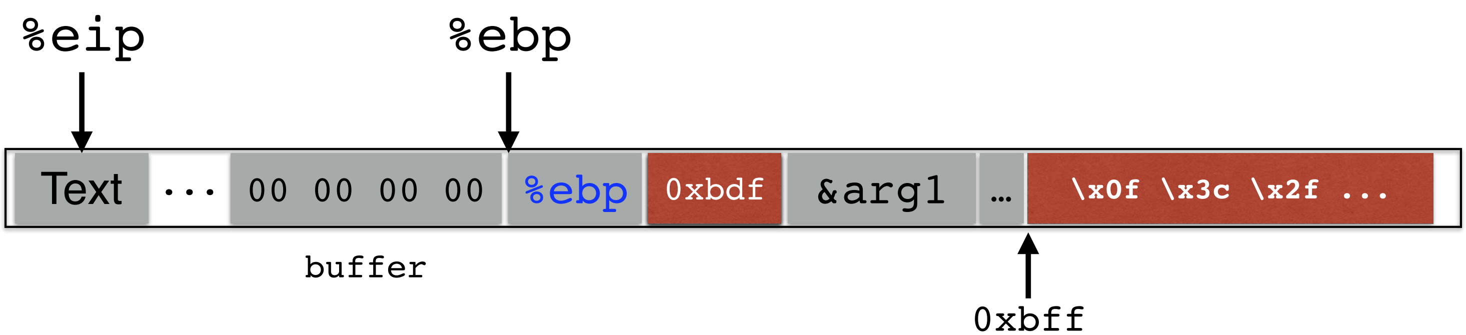
Hijacking the saved `%eip`



But how do we know the address?

Hijacking the saved `%eip`

What if we are wrong?



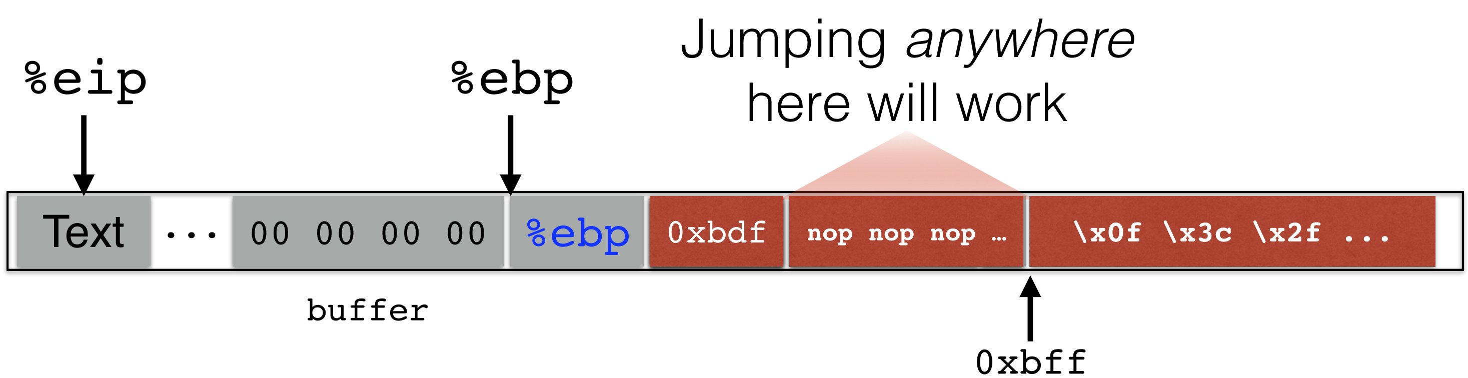
This is most likely data, so the CPU will panic (Invalid Instruction)

Challenge 3
Finding the return address

- If we don't have access to the code, we don't know how far the buffer is from the saved `%ebp`
- One approach: just try a lot of different values!
  - Worst case scenario: it's a 32 (or 64) bit memory space, which means 2^{32} (2^{64}) possible answers
- Without address randomization (discussed later):
  - The **stack always starts** from the same **fixed address**
  - The stack will grow, but usually it **doesn't grow very deeply** (unless the code is heavily recursive)

Improving our chances: `nop` sleds

`nop` is a single-byte instruction
(just moves to the next instruction)



**Now we improve our chances
of guessing by a factor of #nops**

Putting it all together

But it has to be *something*;
we have to start writing wherever
the input to **`gets`**/etc. begins.